OCR Computer Science A Level

1.4.1 Data Types

Advanced Notes

Specification:

1.4.1 a)

- Primitive data types
  - Integer
  - Real / floating point
  - Character
  - String
  - Boolean

1.4.1 b)

- Represent positive integers in binary

1.4.1 c)

- Negative numbers in binary
  - Sign magnitude
  - Two's complement

1.4.1 d)

- Addition and subtraction of binary integers

1.4.1 e)

- Represent positive integers in hexadecimal

1.4.1 f)

- Convert positive integers between binary, hexadecimal and denary

1.4.1 g)

- Representation and normalisation of floating point numbers in binary

1.4.1 h)

- Floating point arithmetic
  - Positive and negative numbers
  - Addition and subtraction

1.4.1 i)

- Bitwise manipulation and masks
  - Shifts
  - Combining with AND, OR, and XOR

1.4.1 j)

- How character sets are used to represent text
  - ASCII
  - Unicode

Data Types

Although data is always stored in binary by computers, the way in which data is represented varies between different types of data. When writing a program, it's essential to make sure data is being stored with the right data type, so that the right operations can be performed on it.

Integer

An integer is a whole number. Integers include zero and negative numbers, they just can't have a fractional part. Integers are useful for counting things.

`6` `47238` `-12` `0` `15`

Real

Real numbers are positive or negative numbers which can, but do not necessarily, have a fractional part. Reals are useful for measuring things. All integers are real numbers. Real numbers can also be represented using floating point, which is explained later.

`0` `-71.5` `5.01` `-80.8` `15`

Character

A character is a single symbol used by a computer. These include the letters A to Z, the numbers 0 to 9 and hundreds of symbols like %, £ and □.

`R` `{` `7` `Σ` `ほ`

String

A string is a collection of characters. While a string can be used to store a single character, they can also be used to store many characters in succession. Strings are useful for storing text and phone numbers which start with a 0, which numeric data types like integers would cut off.

`Hello, world!` `07789`

Synoptic Link

Characters are represented in binary using **character sets**, which are covered later in these notes.

Boolean

Named after the mathematician George Boole (hence written with a capital B), values taken by a Boolean data type are restricted to `True` and `False`. Booleans are useful for recording data that can only take two values, like the state of a power button or whether a line of code has been executed.

```
True                False
```

Representing Positive Integers in Binary

As explained earlier, integers are whole numbers. Computers can store whole numbers using binary. Just like humans count in base 10, computers count in base 2, where each step in place represents a value of two times the previous place.

A single binary digit is called a bit, and eight binary digits can be combined to form a byte. Entertainingly, half a byte (four bits) is called a nybble.

Binary to Decimal

The least significant bit of a binary number is the one furthest to the right, while the most significant bit is furthest to the left. When representing positive integers, the least significant bit always represents a value of 1, with the 2nd least significant bit representing a value of 2, then 4, 8, etc.

8 (2^3)	4 (2^2)	2 (2^1)	1 (2^0)
1	1	0	1

The diagram above shows the place value of each digit, as well as the digit's value (either a 0 or a 1). To work out what the number is, multiply the digit by its place value and add to a total.

For the diagram above, we have **$(8\times1)+(4\times1)+(2\times0)+(1\times1) = 13$** so the binary 1101 is 13 in decimal.

Decimal to Binary

If you have a decimal (denary) number to convert into binary, the first step is to find the largest power of two which is smaller than the number you're converting. Then write out place values in powers of two up to this power.

For example, if we were converting the decimal 47 into binary, we would write out place values up to 32.

32	16	8	4	2	1

Now we need to place a 1 or a 0 in each position so that the total adds up to 47. Starting from the most significant bit (left hand side) we write a 1 if the place value is less than or equal to our value and a 0 otherwise. If we write a 1, then we subtract the place value from our value and use the result for the next stage.

For example: the most significant bit has a value of 32, which is less than 47. Therefore, we write a 1 under 32 and subtract 32 from 47 giving us a new value of 15.

32	16	8	4	2	1
1					

We now look at the next most significant bit, and follow the same steps as before. This time the bit represents 16 and our value is 15. The bit value is greater than our value and so we place a 0.

32	16	8	4	2	1
1	0				

Next up is 8, smaller than 15. Therefore we place a 1 and our new value is 15-8 = 7.

32	16	8	4	2	1
1	0	1			

The next most significant bit is 4. 4 is smaller than 7 and so we place a 1. Our new value is `7-4 = 3`.

32	16	8	4	2	1
1	0	1	1		

Next up is 2, again smaller than our value of 3. We place a one and our new value is 1.

32	16	8	4	2	1
1	0	1	1	1	

Finally, the bit represents 1 and our value is 1, so we place a 1.

32	16	8	4	2	1
1	0	1	1	1	1

We now have that the binary `101111` represents the decimal number 47. You can check your conversion using the method explained earlier for converting from binary to decimal.

It's not unusual to see binary numbers represented as a whole number of bytes (a multiple of eight bits) by adding leading zeros. This does not affect the value of the number. To be represented as a byte, 47 would be written as `00101111`, with two leading `0`s.

Binary Addition

When adding binary, there are four simple rules to remember:

```
1. 0 + 0 + 0 =  0
2. 0 + 0 + 1 =  1
3. 0 + 1 + 1 = 10
4. 1 + 1 + 1 = 11
```

<u>Example:</u> Add the binary numbers `1011` and `1110`.

$$\begin{array}{ccccc} & 1 & 0 & 1 & 1 \\ + & 1 & 1 & 1 & 0 \\ \hline \end{array}$$

Place the two binary numbers above each other so that the digits line up.

$$\begin{array}{ccccc} & 1 & 0 & 1 & 1 \\ + & 1 & 1 & 1 & 0 \\ \hline & & & & 1 \end{array}$$

Starting from the least significant bits (the right hand side), add the values in each column and place the total below. For the first column (highlighted), rule 2 from above applies.

$$\begin{array}{ccccc} & 1 & 0 & 1 & 1 \\ + & 1 & 1 & 1 & 0 \\ \hline & & {}^{1} & 0 & 1 \end{array}$$

Move on to the next column. This time rule 3 applies. In the case that the result of addition for a single column is more than one digit, place the first digit of the result in small writing under the next most significant bit's column.

$$\begin{array}{ccccc} & 1 & 0 & 1 & 1 \\ + & 1 & 1 & 1 & 0 \\ \hline & {}^{1} & 0^{1} & 0 & 1 \end{array}$$

On to the next column, where there is a `0`, a `1` and a small `1`. In this case, rule 3 applies again. Therefore the result is `10`. Because `10` is two digits long, the `1` is written in small writing under the next most significant bit's column.

$$\begin{array}{ccccc} & 1 & 0 & 1 & 1 \\ + & 1 & 1 & 1 & 0 \\ \hline 1 & 1^{1} & 0^{1} & 0 & 1 \end{array}$$

Moving on to the most significant column where there are three `1`s. Rule 4 applies, so the result for this column is `11`. The first digit of the result is written under the next most significant bit's column, but it can be written full size as there are no more columns to add.

$$\begin{array}{ccccc} 1 & 1 & 0 & 0 & 1 \end{array}$$

Finally, the result is read off from the full size numbers at the bottom of each column. In this case, `1011 + 1110 = 11001`.

After carrying out binary addition, it's a good idea to check your answer by converting to decimal if you have time.

Negative Numbers in Binary

So far we've covered conversion between decimal and binary for positive integers. However, binary can represent negative numbers using a few different methods. These methods set out rules for how a bit string should be treated, giving a special meaning to certain bits which allows for the representation of negative numbers.

Sign Magnitude
The most basic way to represent negative numbers in binary is called sign magnitude representation. This is the equivalent of adding a + or - sign in front of a number. However, binary can only use 0s and 1s, so we have to somehow represent + and - using 0 and 1.

The convention used is that a leading 1 is added for a negative number, and a leading 0 is added for a positive number.

For example, the binary 10101101 represents the decimal number 173. Converting to sign magnitude means adding a 0, or to represent -173, add a leading 1.

Binary 173	Sign Magnitude +173	Sign Magnitude -173
10101101	010101101	110101101

Converting from sign magnitude to decimal is as simple as making a note of the most significant bit, remembering the sign and discarding the leading bit. Then convert the remaining bits to decimal using the method explained earlier and add the sign.

For example, the sign magnitude number 101101001 is negative, because it starts with a 1. Remove the 1 and we're left with 01101001 which is 105 in decimal. Add on the minus sign and we have our result: -105.

Two's Complement
Another method of representing negative numbers in binary, two's complement has the added advantage of making binary arithmetic with negative numbers much more simple.

Two's complement works by making the most significant bit negative. For example, with eight bits (a byte) the most significant bit, usually 128, represents -128.

Converting to two's complement is as simple as flipping all of the bits in the positive version of a binary number and adding one. For example, the binary byte representing 7 is 00000111. Flip all the bits and you get 11111000, adding one gives us 11111001.

Writing in the bit values, we can see how this works.

-128	64	32	16	8	4	2	1
1	1	1	1	1	0	0	1

Calculating the equivalent in decimal, using the method explained earlier, we have:

$$(-128\times1)+(64\times1)+(32\times1)+(16\times1)+(8\times1)+(4\times0)+(2\times0)+(1\times1) = \mathbf{-7}$$

Subtracting in Binary using Two's Complement

Two's complement makes subtraction in binary easy. Subtracting a number from another is the same as adding a negative number. This is how binary subtraction works.

Example: Subtract 12 from 8.

	-16	8	4	2	1
	0	1	0	0	0
+	1	0	1	0	0
	1	1	1	0	0

In five bit two's complement, 8 is `01000` and -12 is `10100`. Five is the minimum number of bits required in order to represent `-12`.

The two's complement numbers are then added using the same technique for adding that was explained earlier before the result can be read off as `11100`.

Checking the result, $-16 + 8 + 4 = -4$ so the calculation is correct.

Hexadecimal

In the same way that decimal is base 10, and binary is base 2, hexadecimal is base 16. In addition to the numbers 0-9, hexadecimal makes use of the characters A-F to represent 10-15.

Decimal	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Hexadecimal	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**A**	**B**	**C**	**D**	**E**	**F**

Just like binary, place values in hexadecimal start with 1 (16^0) and go up in powers of 16.

For example, the hexadecimal number 4E7F = 20095

4096 (16^3)	256 (16^2)	16 (16^1)	1 (16^0)
4	E	7	F

Remembering that E represents 14 in decimal and that F represents 15, we have:

$$(4\times4096)+(14\times256)+(7\times16)+(15\times1) = 20095$$

Converting from hexadecimal to binary

To convert hexadecimal to binary, first convert each hexadecimal digit to a decimal digit and then to a binary nybble before combining the nybbles to form a single binary number.

Split into hexadecimal digits

$$\begin{tikzcd} & \text{B2} \arrow[dl] \arrow[dr] & \\ \text{B} & & 2 \end{tikzcd}$$

Convert hexadecimal to decimal

11 2

Convert decimal to binary nybbles

Combine binary nybbles

$$\begin{tikzcd} 1011 \arrow[dr] & & 0010 \arrow[dl] \\ & 10110010 & \end{tikzcd}$$

128	64	32	16	8	4	2	1
1	0	1	1	0	0	1	0

128 + 32 + 16 + 2 = 178

B2 = 178

Converting from hexadecimal to decimal
One way to convert from hexadecimal to decimal is to first convert to binary, as explained above, and then convert from binary to decimal. Alternatively, use the place values of hexadecimal to convert directly to decimal.

256 (16^2)	16 (16^1)	1 (16^0)
4	C	3

For example, 4C3 in hexadecimal is **`(4×256)+(12×16)+(3×1) = 1219`**

Floating Point Numbers in Binary

You can think of floating point binary as being like scientific notation. Take the following example:

$$6.67 \times 10^{-11}$$

Floating point numbers can be split into two parts: mantissa and exponent. In this case, the mantissa is 6.67 and the exponent is -11. When combined, the mantissa and exponent provide all the information needed to work out the actual value being represented.

In this case, the scientific notation represents the value 0.0000000000667. The value 6.67 is shifted 11 times from the decimal point.

We can do the same in binary, provided that we include information about the size of the mantissa and exponent. We also dedicate a single bit to the sign - whether a number is positive or negative.

Take for example a structure with a leading sign bit, a 10-bit mantissa and a 6-bit two's complement exponent.

```
 S                 M                          E

 0    1  1  0  0  1  0  0  1  1  1    0  0  0  1  0  1
```

Just as before, a sign of `0` represents a positive number and a `1` represents a negative number.

The mantissa is always taken to have the binary point (the equivalent of a decimal point, but in binary) after the most significant bit. So this mantissa is actually `1.100100111`.

Next we convert the exponent to decimal using the method explained earlier. In this case, the exponent is 5.

Combining the three parts, we need to move the binary point five places to the right, giving us 110010.0111. We can then convert this to decimal as follows:

32	16	8	4	2	1		0.5	0.25	0.125	0.0625
1	1	0	0	1	0	.	0	1	1	1

32 + 16 + 2 + 0.25 + 0.125 + 0.0625 = **50.4375**

Second example

As before, we're using a format with a single sign bit, a 10-bit mantissa and a 6-bit two's complement exponent.

S					M									E		
1	0	1	0	1	1	0	1	0	0	0	1	1	1	1	0	1

The sign is 1 and so we're dealing with a negative number. The mantissa is 0.101101000 and the exponent is (remembering it's in two's complement) -32 + 16 + 8 + 4 + 1 = -3.

We move the binary point from between the two most significant bits of the mantissa three places to the left, giving us 0.000101101.

1		$\frac{1}{2}$	$\frac{1}{4}$	$\frac{1}{8}$	$\frac{1}{16}$	$\frac{1}{32}$	$\frac{1}{64}$	$\frac{1}{128}$	$\frac{1}{256}$	$\frac{1}{512}$
0	.	0	0	0	1	0	1	1	0	1

Which is $\frac{45}{512}$ in base 10.

Normalisation

Floating point numbers are normalised to make sure that they are as precise as possible in a given number of bits. This essentially equates to making as much use of the mantissa as possible. To normalise a binary number, adjust it so that it starts `01` for a positive number of `10` for a negative number.

Example: Normalise the binary number `000110100101` which is a floating point number with an eight bit mantissa and a four bit exponent.

First, split the number into mantissa and exponent.

0	0	0	1	1	0	1	0	0	1	0	1
Mantissa								Exponent			

Next, adjust the mantissa so that it starts `01` or `10`. In this case, because we're dealing with a positive number, we will move all of the bits two places to the left and add zeros to the end of the mantissa. Our new mantissa is `01101000`.

Because we've made the mantissa bigger by shifting the bits two positions to the left, we must reduce the exponent by two so as to ensure the same number is still represented. The current exponent is 5_{10} so, subtracting two, the new exponent must be 3_{10} which is 0011_2 in binary.

0	1	1	0	1	0	0	0	0	0	1	1
Mantissa								Exponent			

We now have a mantissa that starts with the digits `01`. A positive normalised number.

Addition and Subtraction of Floating Point Numbers

Addition

In order to add floating point binary numbers, their exponents need to be the same. Then it's just a case of adding the mantissas and normalising the result if required.

If the exponents are not the same, the numbers must first be modified so that they have the same exponent. Take the two floating point numbers, each with a single sign bit, a six-bit mantissa and a four-bit exponent below for example.

```
0 000100 0011
0 000101 0010
```

The exponents are not the same, so one must be modified to match the other. The first mantissa needs to be shifted by three, and the second by two. If we shift the first mantissa by one then it will only need shifting by another two, meaning its exponent will match the second.

Carrying this out gives us:

```
0 001000 0010
0 000101 0010
```

Now that the exponents are the same, we can carry out binary addition on the mantissas, like so:

```
    0   0   1   0   0   0
+   0   0   0   1   0   1
   ________________________
    0   0   1   1   0   1
```

Giving us the result 0 001101 0010. However, we're not finished yet. The result still needs to be normalised. To do this, we shift the mantissa one bit to the left and decrease the exponent by one, resulting in `0 011010 0001`.

You can always check addition by converting each of the numbers to decimal and checking the result.

Subtraction

Just like integer subtraction in binary, floating point subtraction involves converting to two's complement and adding.

The first stage is the same as for floating point addition: make the exponents the same. Following this, the mantissa of the number to be subtracted must be converted to two's complement. This is performed by flipping all the bits and adding one.

Now binary addition is carried out on the two numbers, before the result is normalised.

Bitwise Manipulation and Masks

Shifts

A shift performed on binary numbers is called a logical shift. There are two varieties: logical shift left and logical shift right. A shift involves moving all of the bits in a binary number a specified number of places to the right or to the left. This can be thought of as adding a number of leading or trailing zeros.

For example, perform a logical shift left by three places to the binary 10010110
A logical shift left by three places is the same as adding three trailing zeros.
The result is therefore 10010110000.

The result of a logical shift is a multiplication (or division if shifting right) by two to the power of the number of places shifted. The example above has the effect of multiplying the original number by $2^3 = 8$. A logical shift left by one place has the effect of doubling the initial number.

Masks

A mask can be applied to binary numbers by combining them with a logic gate. Take for example the binary numbers 00101011 10111011 masked using the AND gate.

	0	0	1	0	1	0	1	1
AND	1	0	1	1	1	0	1	1
	0	0	1	0	1	0	1	1

Synoptic Link

Logical operations like AND, OR and XOR are covered in more detail in **1.4.3 Boolean Algebra**

AND has the effect of multiplying inputs, so only the digits in the two numbers which are both 1 result in a 1 after the mask has been applied, otherwise the result is 0.

The following diagrams show the output of a mask with OR and XOR on the same inputs.

```
        0   0   1   0   1   0   1   1
 OR     1   0   1   1   1   0   1   1
-------------------------------------
        1   0   1   1   1   0   1   1

        0   0   1   0   1   0   1   1
XOR     1   0   1   1   1   0   1   1
-------------------------------------
        1   0   0   1   0   0   0   0
```

Character Sets for Representing Text

A character set is a published collection of codes and corresponding characters which can be used by computers for representing text.

Two widely used character sets are ASCII and Unicode.

ASCII
Standing for American Standard Code for Information Interchange, ASCII was the leading character set before Unicode. ASCII uses 7 bits to represent $2^7 = 128$ different characters. The capital letters A-Z are represented by codes 65-90 while the lower case letters a-z correspond to codes 97-122. There are also codes for numbers and symbols.

While 128 characters is plenty for standard letters, numbers and symbols, ASCII soon came into trouble when computers needed to represent other languages with different characters.

Unicode
Unicode solves the problem of ASCII's limited character set. Unicode uses a varying number of bits allowing for over 1 million different characters, many of which have yet to be allocated. Because of this, Unicode has enough capacity to represent a wealth of different languages, symbols and emoji.